REMARKS

This After Final Amendment is submitted in response to the final Office Action of October 3, 2006 (hereinafter "the Office Action"). Claims 1, 5, 6, 8, 9, and 22-24 remain pending.

Amendment

Applicants respectfully request entry of this After Final Amendment in accordance with 37 C.F.R. § 1.116 to place this application in condition for allowance.

Claims 1 and 22 are amended to positively recite that the XML document is downloaded into a wireless or hand held mobile device, and that the compiled DTD is executed on a processor of the device. This feature is supported by the specification as originally filed, e.g., in the first four paragraphs of the "Detailed Description" portion of the Application.

Rejection under 35 U.S.C. §101

Applicant notes with appreciation the withdrawal of the rejection under 35 U.S.C. §101 made in the Office Action of May 19, 2005.

Rejections based on prior art

Claims 1 and 22 stand rejected under 35 U.S.C. § 102(e) for being anticipated by published U.S. patent application 10/452,282 (Publication No. 2004/0002952) filed by Lee et al, hereinafter referred to as "Lee." Claims 5, 6, 8, 9, 23, and 24 stand rejected under 35 U.S.C. § 103(a) for being unpatentable over Lee in view of U.S. Patent Application 09/753,038 (Publication No. 2001/0054172) filed by Tuatini, hereinafter referred to as "Tuatini." Applicant respectfully traverses because the prior art references do not show or suggest each and every feature set forth in the claims.

By the present Amendment, claims 1 and 22 now positively recite a step of downloading the compiled DTD into a wireless or handheld mobile device. For the reasons explained below, Lee does not disclose a compiled DTD. Tuatini describes a serialization technique that includes a validation component, but is directed toward a service provider (see, e.g., Tuatini, Figure 5 and corresponding text in paragraph 22). There is no suggestion of

using a compiled DTD in a wireless or hand held mobile device as set forth in the presently amended claims.

The Office Action appears to interpret the term, "Document Type Definition" or "DTD" in an overly broad manner. Applicant respectfully points out that, during examination, the Office is obliged to give claim elements the broadest *reasonable* interpretation to claim elements, and this interpretation must be consistent with the interpretation that those skilled in the art would reach. The "broadest reasonable interpretation" should be the "plain meaning" or "ordinary and customary interpretation" of the term, which may be evidenced by a variety of sources, including: the claims themselves, dictionaries and treatises, the written description, the drawings, and the prosecution history. See MPEP 2111.

In the present Application, the phrase, "Document Type Definition" (hereinafter, "DTD") has a specific meaning in the art. As defined in Applicant's own disclosure, A DTD "specif[ies] the valid information and arrangement of information in the complex XML document" (page 1, lines 26-28). Lee states: "the DTD is the definition of language. Thus, the DTD has to define the structure of an XML document" (paragraph 19). The second reference (Tuatini) cited by the Examiner similarly states: "The DTD's of a document provide meta data that is used by a parser when parsing the document. The meta data includes allowed sequence and nesting of tags, attribute values, names of external files that may be referenced, the formats of external data that may be referenced, and entities that may be encountered" (paragraph 11). The Article "Data Models" presented in *Computer Science Handbook, Second Ed.,* by Allen B. Tucker, which is attached hereto, states, "The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document. However, the DTD does not, in fact, constrain types in the sense of basic types like integer or string. Instead, it only constrains the appearance of subelements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements appear within an element" (p. 52-16, bottom paragraph).

Given that the Applicant, the references relied upon by the Office, and the *Computer Science Handbook* all generally agree on the meaning of the term, DTD, Applicant respectfully submits that the term should be given weight and meaning by the Office. However, the Office Action's statement on page 6, that "Therefore Lee clearly teaches a compiled DTD" implies an *unreasonably broad* interpretation of "DTD." Applicant generally

agrees with the Examiner's understanding of Lee on page 6 of the Office Action, i.e., that Lee teaches an XML validator that receives a DTD document corresponding to an XML document and that the XML validator then applies the XML document to the DTD to verify the validity of the XML document. However, Applicant does not agree with equating the XML validator of Lee to the term "compiled DTD."

Applicant respectfully submits that it is improper to equate "XML validator" of Lee to the claimed "compiled DTD" because the XML validator is not a DTD, as the term would be understood according to its plain meaning given its *broadest reasonable interpretation.* As mentioned above, the term DTD has a specific meaning that is well understood and agreed upon by those skilled in the art, including the primary references relied upon by the Office Action, the present disclosure, and the Encyclopedic *Computer Science Handbook*, as described above. Specifically, that a DTD contains a list of rules and constrains the format of an XML document. It is clear, however, that Lee's XML validator does not contain a list of rules defining elements within an XML document. While the XML validator does receive and parse the DTD, the XML validator itself the DTD does not become a part of the XML validator, since the XML validator is separate and distinct from the data upon which it acts. To call the XML validator a compiled DTD is the equivalent of calling the word processing program upon which this Request for Reconsideration is being drafted, a "Compiled Request for Reconsideration."

Claim 1 specifically recites, *inter alia*: "accessing a compiled document type definition (DTD) for the XML document, the compiled document type definition being executable program code; and verifying the XML document using the compiled DTD, the verifying comprising generating one of a verified XML output or an error" (claim 1, lines 3-7). Claim 22 includes similar language, but is drawn to a machine-readable medium. For the reasons explained above, Applicant respectfully asserts that Lee does not disclose a compiled DTD. The DTD described in Lee is not an executable program, and the XML verifier, which is an executable program, is not a DTD. Since Lee fails to disclose each and every reference set forth in claims 1 and 22, Applicant respectfully submits that claim 1 is not anticipated by Lee, and that the outstanding rejection should be withdrawn.

Since none of the references teach or suggest downloading a compiled DTD into a wireless or handheld mobile device as now claimed, Applicant respectfully submits that the claims should be allowed.

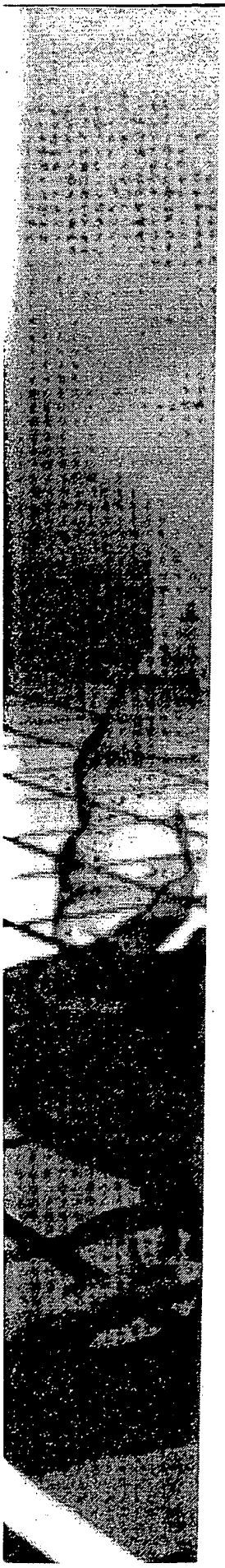
Applicants respectfully submit that the present Application is now in condition for allowance. A Notice of Allowance is therefore respectfully requested.

If the Examiner has any questions concerning the present amendment, the Examiner is kindly requested to contact the undersigned at (408) 774-6933. If any other fees are due in connection with filing this amendment, the Commissioner is also authorized to charge Deposit Account No. 50-0805 (Order No. SUNMP365). A duplicate copy of the transmittal is enclosed for this purpose.

Respectfully submitted,
MARTINE PENILLA & GENCARELLA, LLP

Leonard Heyman
Reg. No. 40, 418

710 Lakeway Drive, Suite 200
Sunnyvale, CA 94085
Telephone: (408) 749-6900
Facsimile: (408) 749-6901
Customer Number 32291



SECOND EDITION

COMPUTER SCIENCE HANDBOOK

EDITOR-IN-CHIEF
ALLEN B. TUCKER

CHAPMAN & HALL/CRC

Published in Cooperation with ACM, The Association for Computing Machinery

Library of Congress Cataloging-in-Publication Data

Computer science handbook / editor-in-chief, Allen B. Tucker—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-58488-360-X (alk. paper)
1.Computer science-Handbooks, manuals, etc. 2. Engineering—Hanbooks, manuals, etc.
I. Tucker, Allen B.

QA76.C54755 2004
004—dc22 2003068758

Visit the CRC Press Web site at www.crcpress.com

International Standard Book Number 1-58488-360-X
Library of Congress Card Number 2003068758
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

52

Data Models

Avi Silberschatz
Yale University

Henry F. Korth
Lehigh University

S. Sudarshan
IIT Bombay

52.1 Introduction

Underlying the structure of a database is the concept of a *data model*. A data model is a collection of conceptual tools for describing the real-world entities to be modeled in the database and the relationships among these entities. Data models differ in the primitives available for describing data and in the amount of semantic detail that can be expressed.

The various data models that have been proposed fall into three different groups: physical data models, record-based logical models, and object-based logical models. Physical data models are used to describe data at the lowest level. Physical data models capture aspects of database system implementation that are not covered in this article. Database system interfaces used by application programs are based on the logical data model; databases hide the underlying implementation details from applications.

This chapter focuses on logical data models, covering the relational data model, the E-R model, the object-oriented and object-relational data models, and XML.

52.2 The Relational Model

The **relational model** is currently the primary data model for commercial data-processing applications. It has attained its primary position because of its simplicity, which eases the job of the programmer, as compared to earlier data models.

A relational database consists of a collection of **tables**, each of which is assigned a unique name. An *instance* of a table storing customer information is shown in Table 52.1. The table has several rows, one for each customer, and several columns, each storing some information about the customer. The values in the *customer-id* column of the *customer* table serve to uniquely identify customers, while other columns store information such as the name, street address, and city of the customer.

The information stored in a database is broken up into multiple tables, each storing a particular kind of information. For example, information about accounts and loans at a bank would be stored in separate tables. Table 52.2 shows an instance of the *loan* table, which stores information about loans taken from the bank.

1-58488-360-X/$0.00+$1.50

TABLE 52.1 The *Customer* Table

Customer-id	*Customer-Name*	*Customer-Street*	*Customer-City*
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

TABLE 52.2 The *Loan* Table

Loan-Number	*Amount*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

TABLE 52.3 The *Borrower* Table

Customer-id	*Loan-Number*
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

In addition to information about "entities" such as customers or loans, there is also a need to store information about "relationships" between such entities. For example, the bank needs to track the relationship between customers and loans. Table 52.3 shows the *borrower* table, which stores information indicating which customers have taken which loans. If several people have jointly taken a loan, the same loan number would appear several times in the table with different customer-ids (e.g., loan number L-17). Similarly, if a particular customer has taken multiple loans, there would be several rows in the table with the customer-id of that customer (e.g., 019-28-3746), with different loan numbers.

52.2.1 Formal Basis

The power of the relational data model lies in its rigorous mathematical foundations and a simple user-level paradigm for representing data. Mathematically speaking, a **relation** is a subset of the Cartesian product of an ordered list of domains. For example, let E be the set of all employee identification numbers, D the set of all department names, and S the set of all salaries. An employment relation is a set of 3-tuples (e, d, s) where $e \in E$, $d \in D$, and $s \in S$. A tuple (e, d, s) represents the fact that employee e works in department d and earns salary s.

At the user level, a relation is represented as a table. The table has one column for each domain and one row for each tuple. Each column has a name, which serves as a column header, and is called an **attribute** of the relation. The list of attributes for a relation is called the **relation schema**. The terms "table" and "relation" are used synonymously, as are row and tuple, as also column and attribute.

Data models also permit the definition of *constraints* on the data stored in the database. For instance, *key constraints* are defined as follows. If a set of attributes L is specified to be a *super-key* for relation r, in any consistent ("legal") database, the set of attributes L would uniquely identify a tuple in r; that is, no two tuples in r can have the same values for all attributes in L. For instance, *customer-id* would form a super-key for relation *customer*. A relation can have more than one super-key, and usually one of the super-keys is chosen as a *primary key*; this key must be a minimal set, that is, dropping any attribute from the set would make it cease to be a super-key.

Another form of constraint is the *foreign key* constraint, which specifies that for each tuple in one relation, there must exist a matching tuple in another relation. For example, a foreign key constraint *from borrower referencing customer* specifies that for each tuple in *borrower*, there must be a tuple in *customer* with a matching *customer-id* value.

Users of a database system can query the data, insert new data, delete old data, or update the data in the database. Of these tasks, the task of querying the data is usually the most complicated. In the case of the relational data model, because data is stored as tables, a user can query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations.

The tuple relational calculus and the domain relational calculus are nonprocedural languages that represent the basic power required in a relational query language. Both of these languages are based on statements written in mathematical logic. We omit details of these languages.

The relational algebra is a procedural query language that defines several operations, each of which takes one or more relations as input and returns a relation as output. For example:

- The **selection** operation is used to get a subset of tuples from a relation, by specifying a predicate. The selection operation $\sigma_P(r)$ returns the set of tuples of r that satisfy the predicate P.
- The **projection** operation $\Pi_L(r)$ is used to return a relation containing a specified set of attributes L of a relation r, removing the other attributes of r.
- The **union** operation $r \cup s$ returns the union of the tuples in r and s. The **intersection** and **difference** operations are similarly defined.
- The **natural join** operation $\bowtie$ is used to combine information from two relations. For example, the natural join of the relations *loan* and *borrower*, denoted *loan* $\bowtie$ *borrower* would be the relation defined as follows. First match each tuple in *loan* with each tuple in *borrower* that has the same values for the shared attribute *loan-number*; for each pair of matching tuples, the join operation creates a tuple containing all attributes from both tuples; the join result relation is the set of all such tuples.

  For instance, the natural join of the *loan* and *borrower* tables in Tables 52.2 and 52.3 contains tuples (L-17, 1000, 321-12-3123) and (L-17, 1000, 963-96-3963), since the tuple with loan number L-17 in the *loan* table matches two different tuples with loan number L-17 in the *borrower* table.

The relational algebra has other operations as well; for example, operations that can aggregate values from multiple tuples, for example by summing them up, or finding their average.

Because the result of a relational algebra operation is itself a relation, it can be used in further operations. As a result, complex expressions with multiple operations can be defined in the relational algebra.

Among the reasons for the success of the relational model are its basic simplicity, representing all data using just a single notion of tables, as well as its formal foundations in mathematical logic and algebra.

The relational algebra and the relational calculi are terse, formal languages that are inappropriate for casual users of a database system. Commercial database systems have, therefore, used languages with more "syntactic sugar." Queries in these languages can be translated into queries in relational algebra.

52.2.2 SQL

The SQL language has clearly established itself as *the* standard relational database language. The SQL language has a data definition component for specifying schemas, and a data manipulation component for querying data as well as for inserting, deleting, and updating data.

We illustrate some examples of queries and updates in SQL. The following query finds the name of the customer whose *customer-id* is 192-83-7465:

select *customer.customer-name*
from *customer*
where *customer.customer-id* = '192-83-7465'

Queries may involve information from more than one table. For example, the following query finds the amount of all loans owned by the customer with customer-id 019-28-3746:

select *loan.loan-number, loan.amount*
from *borrower, loan*
where *borrower.customer-id* = '019-28-3746' **and**
borrower.loan-number = *loan.loan-number*

If the above query were run on the tables shown earlier, the system would find that the loans L-11 and L-23 are owned by customer 019-28-3746, and would print out the amounts of the two loans, namely 900 and 2000.

The following SQL statement adds an interest of 5% to the loan amount of all loans with amounts greater than 1000.

update *loan*
set *amount* = *amount* * 1.05
where *amount* > 10000

Over the years, there have been several revisions of the SQL standard. The most recent is SQL:1999. QBE and Quel are two other significant query languages. Of these, Quel is no longer in widespread use, while QBE is used only in a few database systems such as Microsoft Access.

52.2.3 Relational Database Design

The process of designing a conceptual level schema for a relational database involves the selection of a set of relational schemas. There are several approaches to relational database design. One approach, which we describe in Section 52.3.1, is to create a model of the enterprise using a higher-level data model, such as the entity-relationship model, and then translate the higher-level model into a relational database design.

Another approach is to directly create a design, consisting of a set of tables and a set of attributes for each table. There are often many possible choices that the database designer might make. A proper balance must be struck among three criteria for a good design:

1. Minimization of redundant data
2. Ability to represent all relevant relationships among data items
3. Ability to test efficiently the data dependencies that require certain attributes to be unique identifiers

To illustrate these criteria for a good design, consider a database of employees, departments, and managers. Let us assume that a department has only one manager, but a manager may manage one or more departments. If we use a single relation *emp-info1*(*employee, department, manager*), then we must repeat the manager of a department once for each employee. Thus we have **redundant** data.

We can avoid redundancy by *decomposing* (breaking up) the above relation into two relations *emp-mgr*(*employee, manager*) and *emp-dept*(*manager, department*). However, consider a manager, Martin, who manages both the sales and the service departments. If Clark works for Martin, we cannot represent the fact that Clark works in the service department but not the sales department. Thus we cannot represent all

relevant relationships among data items using the decomposed relations; such a decomposition is called a *lossy-join decomposition*. If instead, we chose the two relations *emp-dept(employee, department)* and *dept-mgr(department, manager)*, we would avoid this difficulty, and at the same time avoid redundancy. With this decomposition, joining the information in the two relations would give back the information in *emp-info1*; such a decomposition is called a *lossless-join decomposition*.

There are several types of data dependencies. The most important of these are **functional dependencies**. A functional dependency is a constraint that the value of a tuple on one attribute or set of attributes determines its value on another. For example, the constraint that a department has only one manager could be stated as "department functionally determines manager." Because functional dependencies represent facts about the enterprise being modeled, it is important that the system check newly inserted data to ensure no functional dependency is violated (as in the case of a second manager being inserted for some department). Such checks ensure that the update does not make the information in the database inconsistent. The cost of this check depends on the design of the database.

There is a formal theory of relational database design that allows us to construct designs that have minimal redundancy, consistent with meeting the requirements of representing all relevant relationships, and allowing efficient testing of functional dependencies. This theory specifies certain properties that a schema must satisfy, based on functional dependencies. For example, a database design is said to be in a *Boyce-Codd normal form* if it satisfies a certain specified set of properties; there are alternative specifications, for instance the *third normal form*. The process of ensuring that a schema design is in a desired normal form is called **normalization**.

More details can be found in standard textbooks on databases; Ullman [Ull88], provides a detailed coverage of database design theory.

52.2.4 History

The relational model was developed in the late 1960s and early 1970s by E.F. Codd. The 1970s saw the development of several experimental database systems based on the relational model and the emergence of a formal theory to support the design of relational databases. The commercial application of relational databases began in the late 1970s but was limited by the poor performance of early relational systems. During the 1980s numerous commercial relational systems with good performance became available. Simultaneously, simple database systems based loosely on the relational approach were introduced for single-user personal computers. In the latter part of the 1980s, efforts were made to integrate collections of personal computer databases with large mainframe databases.

The relational model has since established itself as the primary data model for commercial data processing applications. Earlier generation database systems were based on the *network data model* or the *hierarchical data model*. Those two older models are tied closely to the data structures underlying the implementation of the database. We omit details of these models because they are now of historical interest only.

52.3 Object-Based Models

The relational model is the most widely used data model at the implementation level; most databases in use around the world are relational databases. However, the relational view of data is often too detailed for conceptual modeling. Data modelers need to work at a higher level of abstraction.

Object-based logical models are used in describing data at the conceptual level. The object-based models use the concepts of **entities** or **objects** and relationships among them rather than the implementation-based concepts of the record-based models. They provide flexible structuring capabilities and allow data constraints to be specified explicitly. Several object-based models are in use; some of the more widely known ones are:

- The entity-relationship model
- The object-oriented model
- The object-relational model

The entity-relationship model has gained acceptance in database design and is widely used in practice. The object-oriented model includes many of the concepts of the entity-relationship model, but represents executable code as well as data. The object-relational data model combines features of the object-oriented data model with the relational data model.

The semantic data model and the the functional data model are two other object-based data models; currently, they are not widely used.

52.3.1 The Entity-Relationship Model

The E-R data model derives from the perception of the world or, more specifically, of a particular enterprise in the world, as consisting of a set of basic objects called *entities*, and *relationships* among these objects. It facilitates database design by allowing the specification of an *enterprise schema*, which represents the overall logical structure of a database. The E-R data model is one of several semantic data models; that is, it attempts to represent the meaning of the data.

52.3.1.1 Basics

There are three basic notions that the E-R data model employs: entity sets, relationship sets, and attributes. An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in the universe is an entity.

Each entity is described by a collection of features, called **attributes**. For example, the attributes *account-number* and *balance* may describe one particular account in a bank, and they form attributes of the *account* entity set. Similarly, attributes *customer-name*, *customer-street* address and *customer-city* may describe a *customer* entity.

The values for some attributes may uniquely identify an entity. For example, the attribute *customer-id* may be used to uniquely identify customers (because it may be possible to have two customers with the same name, street address, and city). A unique customer identifier must be assigned to each customer. In the United States, many enterprises use the social-security number of a person (a unique number the U.S. Government assigns to every person in the United States) as a customer identifier.

An entity may be concrete, such as a person or a book, or it may be abstract, such as a bank account, a holiday, or a concept.

An **entity set** is a set of entities of the same type that share the same properties (attributes). The set of all persons working at a bank, for example, can be defined as the entity set *employee*, and the entity John Smith may be a member of the *employee* entity set. Similarly, the entity set *account* might represent the set of all accounts in a particular bank. A database thus includes a collection of entity sets, each of which contains any number of entities of the same type.

Attributes are descriptive properties possessed by all members of an entity set. The designation of attributes expresses that the database stores similar information concerning each entity in an entity set; however, each entity has its own value for each attribute. Possible attributes of the *employee* entity set are *employee-name*, *employee-id*, and *employee-address*. Possible attributes of the *account* entity set are *account-number* and *account-balance*. For each attribute there is a set of permitted values, called the *domain* (or *value set*) of that attribute. The domain of the attribute *employee-name* might be the set of all text strings of a certain length. Similarly, the domain of attribute *account-number* might be the set of all positive integers.

Entities in an entity set are distinguished based on their attribute values. A set of attributes that suffices to distinguish all entities in an entity set is chosen, and called a **primary key** of the entity set. For the *employee* entity set, *employee-id* could serve as a primary key; the enterprise must ensure that no two people in the enterprise can have the same employee identifier.

A **relationship** is an association among several entities. Thus, an *employee* entity might be related by an *emp-dept* relationship to a *department* entity where that employee entity works. For example, there would be an *emp-dept* relationship between John Smith and the bank's credit department if John Smith worked in that department. Just as all *employee* entities are grouped into an *employee* entity set, all *emp-dept* relationship instances are grouped into a *emp-dept* **relationship set**. A relationship set may also have descriptive attributes. For example, consider a relationship set *depositor* between the *customer* and

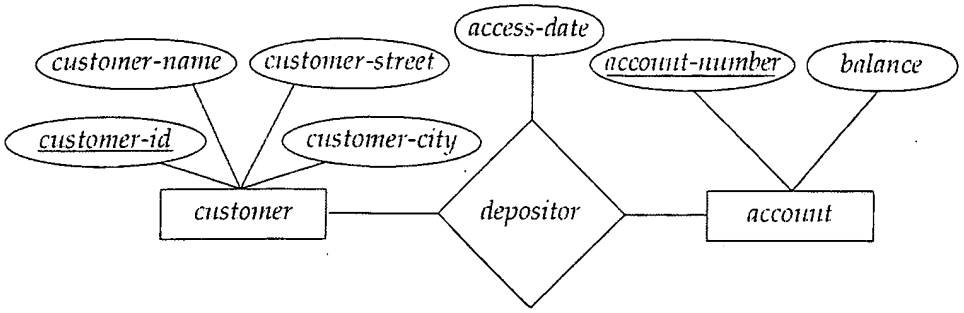


FIGURE 52.1 E-R diagram.

account entity sets. We could associate an attribute *last-access* to specify the date of the most recent access to the account. The relationship sets *emp-dept* and *depositor* are examples of a binary relationship set, that is, one that involves two entity sets. Most of the relationship sets in a database system are binary.

The overall logical structure of a database can be expressed graphically by an E-R **diagram.** Such a diagram consists of the following major components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationship sets
- **Lines**, which link entity sets to relationship sets, and link attributes to both entity sets and relationship sets

An entity-relationship diagram for a portion of our simple banking example is shown in Figure 52.1. The primary key attributes (if any) of an entity set are shown underlined.

Composite attributes are attributes that can be divided into subparts (that is, other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of *first-name*, *middle-initial*, and *last-name*. Using composite attributes in a design schema is a good choice if a user wishes to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions.

The attributes in our examples so far all have a single value for a particular entity. For instance, the *loan-number* attribute for a specific loan entity refers to only one loan number. Such attributes are said to be **single valued**. There may be instances where an attribute has a set of values for a specific entity. Consider an *employee* entity set with the attribute *phone-number*. An employee may have zero, one, or several phone numbers, and hence this type of attribute is said to be **multivalued**.

Suppose that the *customer* entity set has an attribute *age* that indicates the customer's age. If the *customer* entity set also has an attribute *date-of-birth*, we can calculate *age* from *date-of-birth* and the current date. Thus, *age* is a **derived attribute**. The value of a derived attribute is not stored, but is computed when required.

Figure 52.2 shows how composite, multivalued, and derived attributes can be represented in the E-R notation. Ellipses are used to represent composite attributes as well as their subparts, with lines connecting the ellipse representing the attribute to the ellipse representing its subparts. Multivalued attributes are represented using a double ellipse, while derived attributes are represented using a dashed ellipse.

Most of the relationship sets in a database system are *binary*, that is, they involve only two entity sets. Occasionally, however, relationship sets involve more than two entity sets. As an example, consider the entity sets *employee*, *branch*, and *job*. Examples of *job* entities could include manager, teller, auditor, and so on. Job entities may have the attributes *title* and *level*. The relationship set *works-on* among *employee*, *branch*, and *job* is an example of a **ternary relationship**. A ternary relationship among Jones, Perryridge, and manager indicates that Jones acts as a manager at the Perryridge branch. Jones could also act as auditor at the Downtown branch, which would be represented by another relationship. Yet another relationship could be among Smith, Downtown, and teller, indicating Smith acts as a teller at the Downtown branch.

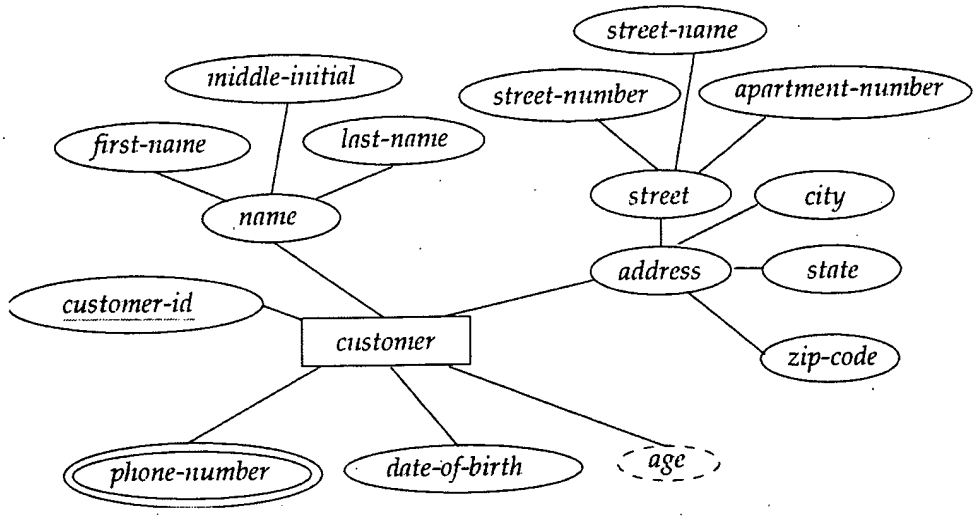


FIGURE 52.2 E-R diagram with composite, multivalued, and derived attributes.

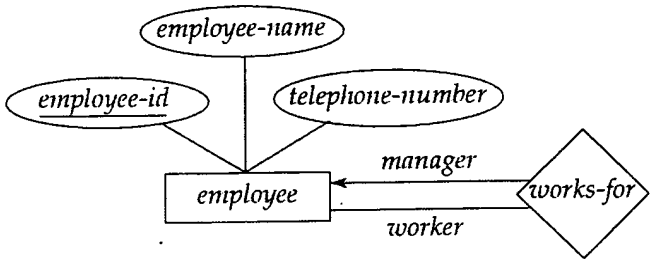


FIGURE 52.3 E-R diagram with role indicators.

Consider, for example, a relationship set *works-for* relating the entity set *employee* with itself. Each employee entity is related to the entity representing the manager of the employee. One employee takes on the role of *worker*, whereas the second takes on the role of *manager*. Roles can be depicted in E-R diagrams as shown in Figure 52.3.

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. Commonly used extended E-R features include specialization, generalization, higher- and lower-level entity sets, attribute inheritance, and aggregation. The notion of specialization and generalization are covered in the context of object-oriented data models in Section 52.3.2. A full explanation of the other features is beyond the scope of this chapter; we refer readers to the references listed at the end of this chapter for additional information.

52.3.1.2 Representing Data Constraints

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set. Therefore, relationships can be classified as many-to-many, many-to-one, or one-to-one. A many-to-many *works-for* relationship between *employee* and *department* exists if a department may have one or more employees and an employee may work for one or more departments. A many-to-one *works-for* relationship between *employee* and *department* exists if a department may have one or more employees but an employee must work for only department. A one-to-one *works-for* relationship exists if a department were required to have exactly one employee,

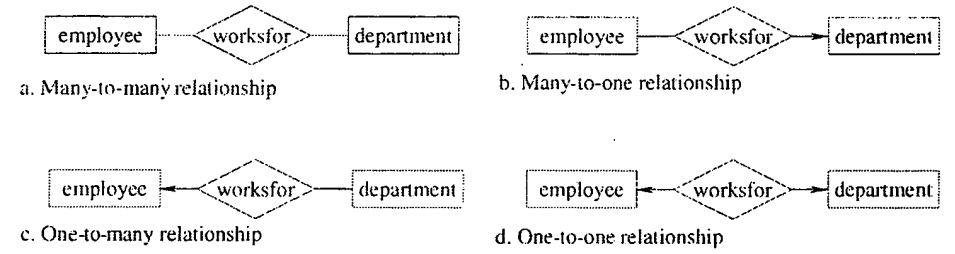


FIGURE 52.4 Relationship cardinalities.

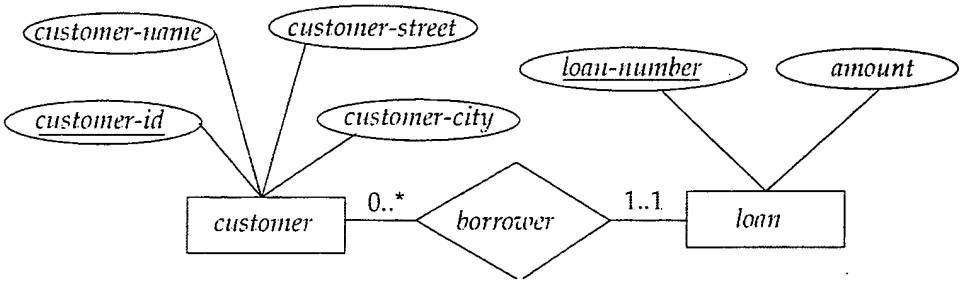


FIGURE 52.5 Cardinality limits on relationship sets.

and an employee was required to work for exactly one department. In an E-R diagram, an arrow is used to indicate the type of relationship, as shown in Figure 52.4.

E-R diagrams also provide a way to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set. An edge between an entity set and a binary relationship set can have an associated minimum and maximum cardinality, shown in the form $l..h$, where l is the minimum and h the maximum cardinality. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value $*$ indicates no limit.

For example, consider Figure 52.5. The edge between *loan* and *borrower* has a cardinality constraint of 1..1, meaning the minimum and the maximum cardinality are both 1. That is, each loan must have exactly one associated customer. The limit 0..$*$ on the edge from *customer* to *borrower* indicates that a customer can have zero or more loans. Thus, the relationship *borrower* is one to many from *customer* to *loan*.

It is easy to misinterpret the 0..$*$ on the edge between *customer* and *borrower*, and think that the relationship *borrower* is many-to-one from *customer* to *loan* — this is exactly the reverse of the correct interpretation.

If both edges from a binary relationship have a maximum value of 1, the relationship is one-to-one. If we had specified a cardinality limit of 1..$*$ on the edge between *customer* and *borrower*, we would be saying that each customer must have at least one loan.

52.3.1.3 Use of E-R Model in Database Design

A high-level data model, such as the E-R model, serves the database designer by providing a conceptual framework in which to specify, in a systematic fashion, the data requirements of the database users and how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to fully characterize the data needs of the prospective database users. The outcome of this phase will be a *specification of user requirements*. The initial specification of user requirements may be based on interviews with the database users, and the designer's own analysis of the enterprise. The description that arises from this design phase serves as the basis for specifying the logical structure of the database.

By applying the concepts of the E-R model, the user requirements are translated into a conceptual schema of the database. The schema developed at this **conceptual design** phase provides a detailed overview of the

enterprise. Stated in terms of the E-R model, the conceptual schema specifies all entity sets, relationship sets, attributes, and mapping constraints. The schema can be reviewed to confirm that all data requirements are indeed satisfied and are not in conflict with each other. The design can also be examined to remove any redundant features. The focus at this point is on describing the data and its relationships, rather than on physical storage details.

A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a *specification of functional requirements*, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. A review of the schema for meeting functional requirements can be made at the conceptual design stage.

The process of moving from a conceptual schema to the actual implementation of the database involves two final design phases. Although these final phases extend beyond the role of data models, we present a brief description of the final mapping from model to physical implementation. In the **logical design** phase, the high-level conceptual schema is mapped onto the implementation data model of the database management system (DBMS). The resulting DBMS-specific database schema is then used in the subsequent **physical design** phase, in which the physical features of the database are specified. These features include the form of file organization and the internal storage structures.

Because the E-R model is extremely useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema, a number of database design tools draw on E-R concepts. Further, the relative simplicity and pictorial clarity of the E-R diagramming technique may well account, in large part, for the widespread use of the E-R model.

52.3.1.4 Deriving a Relational Database Design from the E-R Model

A database that conforms to an E-R diagram can be represented by a collection of tables. For each entity set and each relationship set in the database, there is a unique table that is assigned the name of the corresponding entity set or relationship set. Each table has a number of columns that, again, have unique names. The conversion of database representation from an E-R diagram to a table format is the basis for deriving a relational database design.

The column headers of a table representing an entity set correspond to the attributes of the entity, and the primary key of the entity becomes the primary key of the relation. The column headers of a table representing a relationship set correspond to the primary key attributes of the participating entity sets, and the attributes of the relationship set. Rows in the table can be uniquely identified by the combined primary keys of the participating entity sets. For such a table, the primary keys of the participating entity sets are *foreign keys* of the table. The rows of the tables correspond to individual members of the entity or relationship set.

Table 52.1 through Table 52.3 show instances of tables that correspond, respectively, to the *customer* and *loan* entity sets, the *borrower* relationship set, of Figure 52.5.

52.3.2 Object-Oriented Model

The object-oriented data model is an adaptation of the object-oriented programming paradigm to database systems. The object-oriented approach to programming was first introduced by the language Simula 67, which was designed for programming simulations. More recently, the languages Smalltalk, C++, and Java have become the most widely known object-oriented programming languages. Database applications in such areas as computer-aided design and bio-informatics do not fit the set of assumptions made for older, data-processing-style applications. The object-oriented data model has been proposed to deal with some of these applications. The model is based on the concept of encapsulating data, and code that operates on that data, in an object.

52.3.2.1 Basics

Like the E-R model, the object-oriented model is based on a collection of objects. Entities, in the sense of the E-R model, are represented as **objects** with attribute values represented by *instance variables* within

the object. The value stored in an instance variable may itself be an object. Objects can contain objects to an arbitrarily deep level of nesting. At the bottom of this hierarchy are objects such as integers, character strings, and other data types that are built into the object-oriented system and serve as the foundation of the object-oriented model. The set of built-in object types varies from system to system.

In addition to representing data, objects have the ability to initiate operations. An object may send a *message* to another object, causing that object to execute a *method* in response. Methods are procedures, written in a general-purpose programming language, that manipulate the object's local instance variables and send messages to other objects. Messages provide the only means by which an object can be accessed. Therefore, the internal representation of an object's data need not influence the implementation of any other object. Different objects may respond differently to the same message. This encapsulation of code and data has proven useful in developing higher modular systems. It corresponds to the programming language concept of abstract data types.

The only way in which one object can access the data of another object is by invoking a method of that other object. This is called *sending a message* to the object. Thus, the call interface of the methods of an object defines its externally visible part. The internal part of the object — the instance variables and method code — are not visible externally. The result is two levels of data abstraction.

To illustrate the concept, consider an object representing a bank account. Such an object contains instance variables *account-number* and *account-balance*, representing the account number and account balance. It contains a method *pay-interest*, which adds interest to the balance. Assume that the bank had been paying 4% interest on all accounts but now is changing its policy to pay 3% if the balance is less than $1000 or 4% if the balance is $1000 or greater. Under most data models, this would involve changing code in one or more application programs. Under the object-oriented model, the only change is made within the *pay-interest* method. The external interface to the object remains unchanged.

52.3.2.2 Classes

Objects that contain the same types of values and the same methods are grouped together into *classes*. A class may be viewed as a type definition for objects. This combination of data and code into a type definition is similar to the programming language concept of abstract data types. Thus, all *employee* objects may be grouped into an *employee* class. Classes themselves can be grouped into a hierarchy of classes; for example, the *employee* class and the *customer* classes may be grouped into a *person* class. The class *person* is a **superclass** of the *employee* and *customer* classes because all objects of the *employee* and *customer* classes also belong to the *person* class. Superclasses are also called **generalizations**. Correspondingly, the *employee* and *customer* classes are **subclasses** of *person*; subclasses are also called **specializations**.

The hierarchy of classes allows sharing of common methods. It also allows several distinct views of objects: an employee, for an example, may be viewed either in the role of person or employee, whichever is more appropriate.

52.3.2.3 The Unified Modeling Language UML

The **Unified Modeling Language** (UML) is a standard for creating specifications of various components of a software system. Some of the parts of UML are:

- **Class diagram.** Class diagrams play the same role as E-R diagrams, and are used to model data. Later in this section we illustrate a few features of class diagrams and how they relate to E-R diagrams.
- **Use case diagram.** Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money or registering for a course).
- **Activity diagram.** Activity diagrams depict the flow of tasks between various components of a system.
- **Implementation diagram.** Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

We do not attempt to provide detailed coverage of the different parts of UML here; we only provide some examples illustrating key features of UML class diagrams. See the bibliographic notes for references on UML for more information.

UML class diagrams model objects, whereas E-R models entities. Objects are similar to entities, and have attributes, but additionally provide a set of functions (called methods) that can be invoked to compute values on the basis of attributes of the objects, or to update the object itself. Class diagrams can depict methods in addition to attributes.

We represent binary relationship sets in UML by drawing a line connecting the entity sets. We write the relationship set name adjacent to the line. We may also specify the role played by an entity set in a relationship set by writing the role name on the line adjacent to the entity set. Alternatively, we may write the relationship set name in a box, along with attributes of the relationship set, and connect the box by a dotted line to the line depicting the relationship set. This box can then be treated as an entity set, in the same way as an aggregation in E-R diagrams and can participate in relationships with other entity sets. UML 1.3 supports non-binary relationships, using the same diamond notation used in E-R diagrams.

Cardinality constraints are specified in UML in the same way as in E-R diagrams, in the form $l..h$, where l denotes the minimum and h the maximum number of relationships an entity can participate in. However, the interpretation here is that the constraint indicates the minimum/maximum number of relationships an object can participate in, *given that the other object in the relationship is fixed.* You should be aware that, as a result, the positioning of the constraints is exactly the reverse of the positioning of constraints in E-R diagrams, as shown in Figure 52.6. The constraint $0..*$ on the $E2$ side and $0..1$ on the $E1$ side means that each $E2$ entity can participate in, at most, one relationship, whereas each $E1$ entity can participate in many relationships; in other words, the relationship is many-to-one from $E2$ to $E1$.

Single values such as 1 or $*$ may be written on edges; the single value 1 on an edge is treated as equivalent to $1..1$, while $*$ is equivalent to $0..*$.

We represent generalization and specialization in UML by connecting entity sets by a line with a triangle at the end corresponding to the more general entity set. For instance, the entity set *person* is a generalization of *customer* and *employee*. UML diagrams can also represent explicitly the constraints of disjoint/overlapping on generalizations. For instance, if the *customer*/*employee*-to-*person* generalization is disjoint, it means that no one can be both a *customer* and an *employee*. An overlapping generalization allows a person to be both a *customer* and an *employee*. Figure 52.6 shows how to represent disjoint and overlapping generalizations of *customer* and *employee* to *person*.

52.3.2.4 Object-Oriented Database Programming Languages

There are two approaches to creating an object-oriented database language: the concepts of object orientation can be added to existing database languages, or existing object-oriented languages can be extended to deal with databases by adding concepts such as persistence and collections. Object-relational database systems take the former approach. Persistent programming languages follow the latter approach.

Persistent extensions to C++ and Java have made significant technical progress in the past decade. Several object-oriented database systems succeeded in integrating persistence fairly seamlessly and orthogonally with existing language constructs. The Object Data Management Group (ODMG) developed standards for integrating persistence support into several programming languages such as Smalltalk, C++, and Java. However, object-oriented databases based on persistent programming languages have faced significant hurdles in commercial adoption, in part because of their lack of support for legacy applications, and in part because the features provided by object-oriented databases did not make a significant different to typical data processing applications.

Object-relational database systems, which integrate object-oriented features with traditional relational support, have fared better commercially because they offer an easy upgrade path for existing applications.

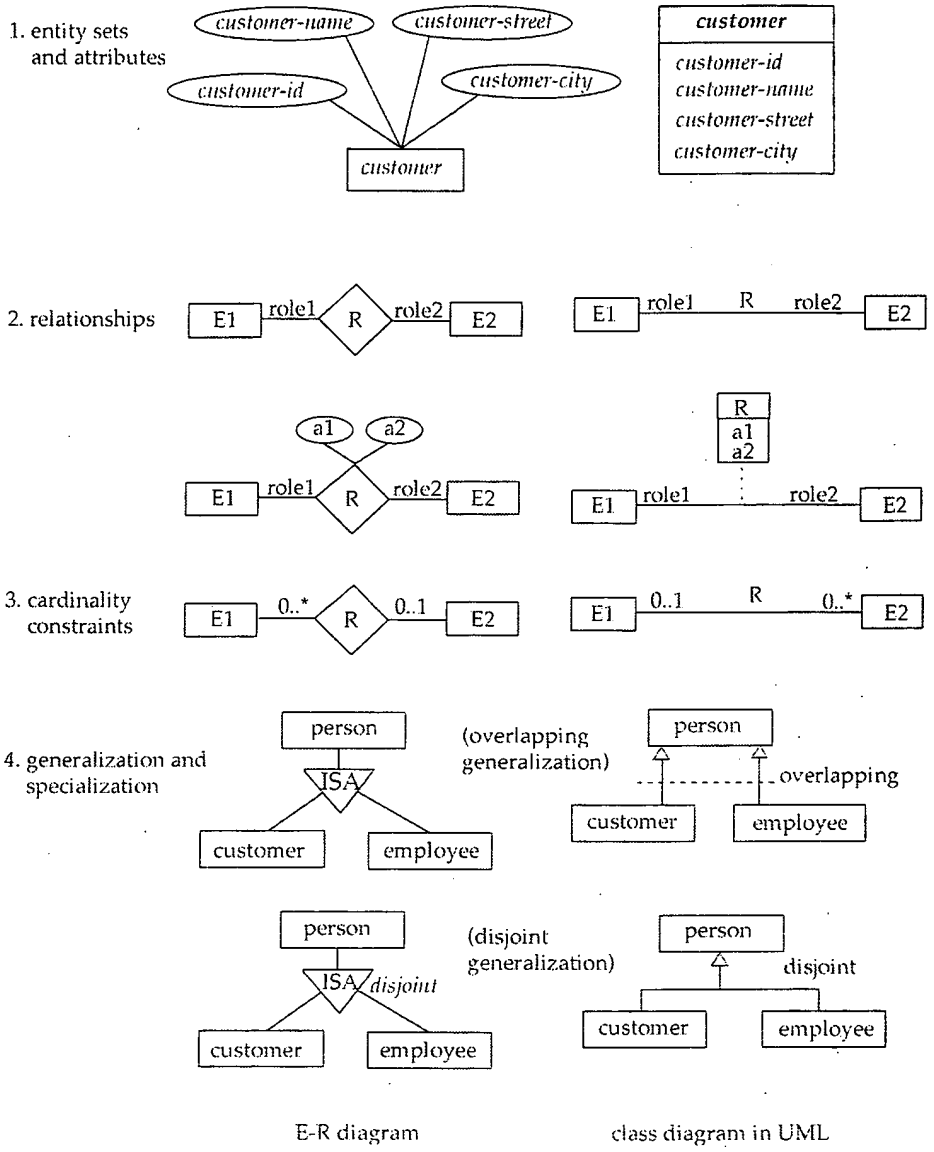


FIGURE 52.6 Correspondence of symbols used in the E-R diagram and UML class diagram notation.

52.3.3 Object-Relational Data Models

Object-relational data models extend the relational data model by providing a richer type system including object orientation. Constructs are added to relational query languages such as SQL to deal with the added data types. The extended type systems allow attributes of tuples to have complex types, including non-atomic values such as nested relations. Such extensions attempt to preserve the relational foundations, in particular the declarative access to data, while extending the modeling power.

Object-relational database systems (that is, database systems based on the object-relational model) provide a convenient migration path for users of relational databases who wish to use object-oriented features. Complex types such as nested relations are useful to model complex data in many applications. Object-relational systems combine complex data based on an extended relational model with object-oriented concepts such as object identity and inheritance. Relations are allowed to form an inheritance hierarchy;

each tuple in a lower-level relation must correspond to a unique tuple in a higher-level relation that represents information about the same object. Inheritance of relations provides a convenient way of modeling roles, where an object can acquire and relinquish roles over a period of time.

Several object-oriented extensions to SQL have been proposed in the recent past. The SQL:1999 standard supports a variety of object-oriented features, including complex data types such as records and arrays, and type hierarchies with classes and subclasses. Values of attributes can be of complex types. Objects, however, do not have an independent existence; they correspond to tuples in a relation. SQL:1999 also supports references to objects; references must be to objects of a particular type, which are stored as tuples of a particular relation.

SQL:1999 supports table inheritance; if r is a subtable of s, the type of tuples of r must be a subtype of the type of tuples of s. Every tuple present in r is implicitly (automatically) present in s as well. A query on s would find all tuples inserted directly to s as well as tuples inserted into r; however, only the attributes of table s would be accessible, even for the r tuples. Thus, subtables can be used to represent specialization/generalization hierarchies. However, while subtables in the SQL:1999 standard can be used to represent disjoint specializations, where an object cannot belong to two different subclasses of a particular class, they cannot be used to represent the general case of overlapping specialization.

In addition to object-oriented data-modeling features, SQL:1999 supports an imperative extension of the SQL query language, providing features such as *for* and *while* loops, *if-then-else* statements, procedures, and functions.

52.4 XML

Unlike most of the data models, the **Extensible Markup Language** (XML) was not originally conceived as a database technology. In fact, like the *Hyper-Text Markup Language* (HTML) on which the World Wide Web is based, XML has its roots in document management. However, unlike HTML, XML can represent database data, as well as many other kinds of structured data used in business applications. It is particularly useful as a data format when an application must communicate with another application, or integrate information from several other applications.

The term **markup** in the context of documents refers to anything in a document that is not intended to be part of the printed output. For the family of markup languages that includes HTML and XML, the markup takes the form of **tags** enclosed in angle-brackets, <>. Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the document to which the tag refers. For example, the title of a document might be marked up as follows:

<title>Database System Concepts</title>

Unlike HTML, XML does not prescribe the set of tags allowed, and the set may be specialized as needed. This feature is the key to XML's major role in data representation and exchange, whereas HTML is used primarily for document formatting.

For example, in our running banking application, account and customer information can be represented as part of an XML document as in Table 52.4. Observe the use of tags such as account and account-number. These tags provide context for each value and allow the semantics of the value to be identified. The contents between a start tag and its corresponding end tag is called an **element**.

Compared to storage of data in a database, the XML representation may be inefficient because tag names are repeated throughout the document. However, despite this disadvantage, an XML representation has significant advantages when it is used to exchange data, for example, as part of a message:

- The presence of the tags makes the message **self-documenting**; that is, a schema need not be consulted to understand the meaning of the text. We can readily read the fragment above, for example.
- The format of the document is not rigid. For example, if some sender adds additional information, such as a tag last-accessed noting the last date on which an account was accessed, the recipient of the XML data may simply ignore the tag. The ability to recognize and ignore unexpected tags allows the format of the data to evolve over time, without invalidating existing applications.

TABLE 52.4 XML Representation of Bank Information

```
<bank>
   <account>
      <account-number> A-101 </account-number>
      <branch-name> Downtown </branch-name>
      <balance> 500 </balance>
   </account>
   <account>
      <account-number> A-102 </account-number>
      <branch-name> Perryridge </branch-name>
      <balance> 400 </balance>
   </account>
   <account>
      <account-number> A-201 </account-number>
      <branch-name> Brighton </branch-name>
      <balance> 900 </balance>
   </account>
   <customer>
      <customer-name> Johnson </customer-name>
      <customer-street> Alma </customer-street>
      <customer-city> Palo Alto </customer-city>
   </customer>
   <customer>
      <customer-name> Hayes </customer-name>
      <customer-street> Main </customer-street>
      <customer-city> Harrison </customer-city>
   </customer>
   <depositor>
      <account-number> A-101 </account-number>
      <customer-name> Johnson </customer-name>
   </depositor>
   <depositor>
      <account-number> A-201 </account-number>
      <customer-name> Johnson </customer-name>
   </depositor>
   <depositor>
      <account-number> A-102 </account-number>
      <customer-name> Hayes </customer-name>
   </depositor>
</bank>
```

- Elements can be nested inside other elements, to any level of nesting. Table 52.5 shows a representation of the bank information from Table 52.4, but with account elements nested within customer elements.

  The nested representation permits representation of complex information within a single document. For instance, a purchase order element may have within it elements representing the supplier, customer, and each of the parts ordered. Each of these elements, in turn, may have subelements; for instance, a part element may have subelements for its part number, name, and price.

  Although the nested representation makes it easier to represent some information, it can result in redundancy; for example, the nested representation of the bank database would store an account element redundantly (multiple times) if it is owned by multiple customers.

  Nested representations are widely used in XML data interchange applications to avoid joins. For instance, a shipping application would store the full address of sender and receiver redundantly on a shipping document associated with each shipment, whereas a normalized representation may require a join of shipping records with a *company-address* relation to get address information.
- Because the XML format is widely accepted, a wide variety of tools are available to assist in its processing, including browser software and database tools.

TABLE 52.5 Nested XML Representation of Bank Information

```
<bank-1>
   <customer>
      <customer-name> Johnson </customer-name>
      <customer-street> Alma </customer-street>
      <customer-city> Palo Alto </customer-city>
      <account>
         <account-number> A-101 </account-number>
         <branch-name> Downtown </branch-name>
         <balance> 500 </balance>
      </account>
      <account>
         <account-number> A-201 </account-number>
         <branch-name> Brighton </branch-name>
         <balance> 900 </balance>
      </account>
   </customer>
   <customer>
      <customer-name> Hayes </customer-name>
      <customer-street> Main </customer-street>
      <customer-city> Harrison </customer-city>
      <account>
         <account-number> A-102 </account-number>
         <branch-name> Perryridge </branch-name>
         <balance> 400 </balance>
      </account>
   </customer>
</bank-1>
```

Just as SQL is the dominant *language* for querying relational data, XML is becoming the dominant *format* for data exchange.

In addition to elements, XML specifies the notion of an *attribute*. For example, the type of an account is represented below as an attribute named acct-type.

```
...
   <account acct-type= "checking" >
      <account-number> A-102 </account-number>
      <branch-name> Perryridge </branch-name>
      <balance> 400 </balance>
   </account>
...
```

The attributes of an element appear as *name=value* pairs before the closing ">" of a tag. Attributes are strings and do not contain markup. Furthermore, an attribute name can appear only once in a given tag, unlike subelements, which may be repeated.

Note that in a document construction context, the distinction between subelement and attribute is important — an attribute is implicitly text that does not appear in the printed or displayed document. However, in database and data exchange applications of XML, this distinction is less relevant, and the choice of representing data as an attribute or a subelement is often arbitrary.

The **document type definition** (DTD) is an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document. However, the DTD does not, in fact, constrain types in the sense of basic types like integer or string. Instead, it only constrains the appearance of subelements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements appear within an element.

For instance, the DTD for the XML data in Table 52.5 is shown below:

```
<!DOCTYPE bank [
   <!ELEMENT bank (customer*)>
   <!ELEMENT customer ( customer-name customer-street customer-city account+)>
   <!ELEMENT customer-name( #PCDATA )>
   <!ELEMENT customer-street( #PCDATA )>
   <!ELEMENT customer-city( #PCDATA )>
   <!ELEMENT account ( account-number branch-name balance )>
   <!ELEMENT account-number ( #PCDATA )>
   <!ELEMENT branch-name ( #PCDATA )>
   <!ELEMENT balance( #PCDATA )>
] >
```

The above DTD indicates that a bank may have zero or more customer subelements. Each customer element has a single occurrence of each of the subelements customer-name, customer-street, and customer-city, and one or more subelements of type account. These subelements customer-name, customer-street, and customer-city are declared to be of type #PCDATA, indicating that they are character strings with no further structure (PCDATA stands for "parsed character data"). Each account element, in turn, has a single occurrence of each of the subelements account-number, branch-name, and balance.

The following DTD illustrates a case where the nesting can be arbitrarily deep; such a situation can arise with complex parts that subparts that themselves have complex subparts, and so on.

```
<!DOCTYPE parts [
   <!ELEMENT part (name, subpartinfo*)>
   <!ELEMENT subpartinfo (part, quantity)>
   <!ELEMENT name ( #PCDATA )>
   <!ELEMENT quantity ( #PCDATA )>
] >
```

The above DTD specifies that a part element may contain within it zero or more subpart elements, each of which in turn contains a part element. DTDs such as the above, where an element type is recursively contained within an element of the same type, are called **recursive DTDs**. The **XMLSchema** language plays the same role as DTDs, but is more powerful in terms of the types and constraints it can specify.

The **XPath** and **XQuery** languages are used to query XML data. The XQuery language can be thought of as an extension of SQL to handle data with nested structure, although its syntax is different from that of SQL.

Many database systems store XML data by mapping them to relations. Unlike in the case of E-R diagram to relation mappings, the XML to relation mappings are more complex and done transparently. Users can write queries directly in terms of the XML structure, using XML query languages.

In summary, the XML language provides a flexible and self-documenting mechanism for modeling data, supporting a variety of features such as nested structures and multivalued attributes, and allowing multiple types of data to be represented in a single document. Although the basic XML model allows data to be arbitrarily structured, the schema of a document can be specified using DTDs or the XMLSchema language. Both these mechanisms allow the schema to be flexibly and partially specified, unlike the rigid schema of relational data, thus supporting **semi-structured data**.

52.5 Further Reading

- **The Relational Model.** The relational model was proposed by E.F. Codd of the IBM San Jose Research Laboratory in the late 1960s [Cod70]. Following Codd's original paper, several research projects were formed with the goal of constructing practical relational database systems, including System R at the IBM San Jose Research Laboratory (Chamberlin et al. [CAB+81]), Ingres at the

University of California at Berkeley (Stonebraker [Sto86b]), and Query-by-Example at the IBM T.J. Watson Research Center (Zloof [Zlo77]).

General discussion of the relational data model appears in most database texts, including Date [Dat00], Ullman [Ull88], Elmasri and Navathe [EN00], Ramakrishnan and Gehrke [RG02], and Silberschatz et al. [SKS02]. Textbook descriptions of the SQL-92 language include Date and Darwen [DD97] and Melton and Simon [MS93].

Textbook descriptions of the network and hierarchical models, which predated the relational model, can be found on the Web site http://www.db-book.com (this is the Web site of the text by Silberschatz et al. [SKS02])

- **The Object-Based Models.**
  - **The Entity-Relationship Model.** The entity-relationship data model was introduced by Chen [Che76]. Basic textbook discussions are offered by Elmasri and Navathe [EN00], Ramakrishnan and Gehrke [RG02], and Silberschatz et al. [SKS02]. Various data manipulation languages for the E-R model have been proposed, although none is in widespread commercial use. The concepts of generalization, specialization, and aggregation were introduced by Smith and Smith [SS77].
  - **Object-Oriented Models.** Numerous object-oriented database systems were implemented as either products or research prototypes. Some of the commercial products include ObjectStore, Ontos, Orion, and Versant. More information on these may be found in overviews of object-oriented database research, such as Kim and Lochovsky [KL89], Zdonik and Maier [ZM90], and Dogac et al. [DOBS94]. The ODMG standard is described by Cattell [Cat00].

    Descriptions of UML may be found in Booch et al. [BJR98] and Fowler and Scott [FS99].
  - **Object-Relational Models.** The nested relational model was introduced in [Mak77] and [JS82]. Design and normalization issues are discussed in [OY87, [RK87], and [MNE96]. POSTGRES ([SR86] and [Sto86a]) was an early implementation of an object-relational system. Commercial databases such as IBM DB2, Informix, and Oracle support various object-relational features of SQL:1999. Refer to the user manuals of these systems for more details.

    Melton et al. [MSG01] and Melton [Mel02] provide descriptions of SQL:1999; [Mel02] emphasizes advanced features, such as the object-relational features, of SQL:1999. Date and Darwen [DD00] describes future directions for data models and database systems.
- **XML.** The XML Cover Pages site (www.oasis-open.org/cover/) contains a wealth of XML information, including tutorial introductions to XML, standards, publications, and software. The World Wide Web Consortium (W3C) acts as the standards body for Web-related standards, including basic XML and all the XML-related languages such as XPath, XSLT, and XQuery. A large number of technical reports defining the XML related standards are available at www.w3c.org.

  A large number of books on XML are available in the market. These include [CSK01], [CRZ03], and [W+00].

Defining Terms

Attribute: 1. A descriptive feature of an entity or relationship in the entity-relationship model. 2. The name of a column header in a table, or, in relational-model terminology, the name of a domain used to define a relation.

Class: A set of objects in the object-oriented model that contains the same types of values and the same methods; also, a type definition for objects.

Data model: A collection of conceptual tools for describing the real-world entities to be modeled in the database and the relationships among these entities.

Element: The contents between a start tag and its corresponding end tag in an XML document.

Entity: A distinguishable item in the real-world enterprise being modeled by a database schema.

Foreign key: A set of attributes in a relation schema whose value identifies a unique tuple in another relational schema.

Functional dependency: A rule stating that given values for some set of attributes, the value for some other set of attributes is uniquely determined. X functionally determines Y if whenever two tuples in a relation have the same value on X, they must also have the same value on Y.

Generalization: A superclass; an entity set that contains all the members of one or more specialized entity sets.

Instance variable: attribute values within objects.

Key: 1. A set of attributes in the entity relationship model that serves as a unique identifier for entities. Also known as *superkey*. 2. A set of attributes in a relation schema that functionally determines the entire schema. 3. *Candidate key*: a minimal key. 4. *Primary key*: a candidate key chosen as the primary means of identifying/accessing an entity set, relationship set, or relation.

Message: The means by which an object invokes a method in another object.

Method: Procedures within an object that operate on the instance variables of the object and/or send messages to other objects.

Normal form: A set of desirable properties of a schema. Examples include the Boyce-Codd normal form and the third normal form.

Object: Data and behavior (methods) representing an entity.

Persistence: The ability of information to survive (persist) despite failures of all kinds, including crashes of programs, operating systems, networks, and hardware.

Relation: 1. A subset of a Cartesian product of domains. 2. Informally, a table.

Relation schema: A type definition for relations, consisting of attribute names and a specification of the corresponding domains.

Relational algebra: An algebra on relations; consists of a set of operations, each of which takes as input one or more relations and returns a relation, and a set of rules for combining operations to create expressions.

Relationship: An association among several entities.

Subclass: A class that lies below some other class (a superclass) in a class inheritance hierarchy; a class that contains a subset of the objects in a superclass.

Subtable: A table such that (a) its tuples are of a type that is a subtype of the type of tuples of another table (the supertable), and (b) each tuple in the subtable has a corresponding tuple in the supertable.

Specialization: A subclass; an entity set that contains a subset of entities of another entity set.

References

[BJR98] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[CAB+81] D.D. Chamberlin, M.M. Astrahan, M.W. Blasgen, J.N. Gray, W.F. King, B.G. Lindsay, R.A. Lorie, J.W. Mehl, T.G. Price, P.G. Selinger, M. Schkolnick, D.R. Slutz, I.L. Traiger, B.W. Wade, and R.A. Yost. A history and evaluation of System R. *Communications of the ACM*, 24(10):632–646, October 1981.

[Cat00] R. Cattell, Editor. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[Che76] P.P. Chen. The Entity-Relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, January 1976.

[Cod70] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6): 377–387, June 1970.

[CRZ03] A.B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.

[CSK01] B. Chang, M. Scardina, and S. Kiritzov. *Oracle9i XML Handbook*. McGraw-Hill, 2001.

[Dat00] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 7th edition, 2000.

[DD97] C.J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 4th edition, 1997.

[DD00] C.J. Date and H. Darwen. *Foundation for Future Database Systems: The Third Manifesto*. Addison Wesley, 2nd edition, 2000.

[DOBS94] A. Dogac, M.T. Ozsu, A. Biliris, and T. Selis. *Advances in Object-Oriented Database Systems*, volume 130. Springer Verlag, 1994. Computer and Systems Sciences, NATO ASI Series F.

[EN00] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, 3rd edition, 2000.

[FS99] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2nd edition, 1999.

[JS82] G. Jaeschke and H.J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 124–138, 1982.

[KL89] W. Kim and F. Lochovsky, Editors. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, 1989.

[Mak77] A. Makinouchi. A consideration of normal form on not-necessarily normalized relations in the relational data model. In *Proc. of the International Conf. on Very Large Databases*, pages 447–453, 1977.

[Mel02] J. Melton. *Advanced SQL: 1999 — Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann, 2002.

[MNE96] W.Y. Mok, Y.-K. Ng, and D.W. Embley. A normal form for precisely characterizing redundancy in nested relations. *ACM Transactions on Database Systems*, 21(1):77–106, March 1996.

[MS93] J. Melton and A.R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[MSG01] J. Melton, A.R. Simon, and J. Gray. *SQL: 1999 — Understanding Relational Language Components*. Morgan Kaufmann, 2001.

[OY87] G. Ozsoyoglu and L. Yuan. Reduced MVDs and minimal covers. *ACM Transactions on Database Systems*, 12(3):377–394, September 1987.

[RG02] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2002.

[RK87] M.A. Roth and H.F. Korth. The design of ¬1nf relational databases into nested normal form. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 143–159, 1987.

[SKS02] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 4th edition, 2002.

[SR86] M. Stonebraker and L. Rowe. The design of postgres. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1986.

[SS77] J.M. Smith and D.C.P. Smith. Database abstractions: aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, March 1977.

[Sto86a] M. Stonebraker. Inclusion of new types in relational database systems. In *Proc. of the International Conf on Data Engineering*, pages 262–269, 1986.

[Sto86b] M. Stonebraker, Editor. *The Ingres Papers*. Addison-Wesley, 1986.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-base Systems*, Volume 1. Computer Science Press, Rockville, MD, 1988.

[W+00] K. Williams (Editor) et al. *Professional XML Databases*. Wrox Press, 2000.

[Zlo77] M.M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

[ZM90] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.